System design: WhatsApp

Suresh Podeti
7 min read · Oct 26, 2023

Introduction

In today's technological world, WhatsApp is an important **messaging application** that connects billions of people around the globe. As per the latest data, WhatsApp has over **2.7 billion** monthly active users in over 180 countries. It is predicted that this number will reach 3.14 billion by 2025.

Requirements

Functional

- **Conversation**: The system should support one-on-one and group conversations between users.
- **Acknowledgment**: The system should support message delivery acknowledgment, such as sent, delivered, and read.
- **Sharing**: The system should support sharing of media files, such as images, videos, and audio.
- **Chat storage**: The system must support the persistent storage of chat messages when a user is offline until the successful delivery of messages.
- **Push notifications**: The system should be able to notify offline users of new messages once their status becomes online.

Non-functional

- **Low latency**: Users should be able to receive messages with low latency.
- **Consistency**: Messages should be delivered in the order they were sent. Moreover, users must see the same chat history on all of their devices.

- **Availability**: The system should be highly available. However, the availability can be compromised in the interest of consistency.
- **Security**: The system must be secure via end-to-end encryption. The end-to-end encryption ensures that only the two communicating parties can see the content of messages. Nobody in between, not even WhatsApp, should have access.

High level design

At an abstract level, the high-level design consists of a chat server responsible for communication between the sender and the receiver. When a user wants to send a message to another user, both connect to the chat server. Both users send their messages to the chat server. The chat server then sends the message to the other intended user and also stores the message in the database.

Note: Sender and receiver may not necessarily connected to same chat server / Websocket server

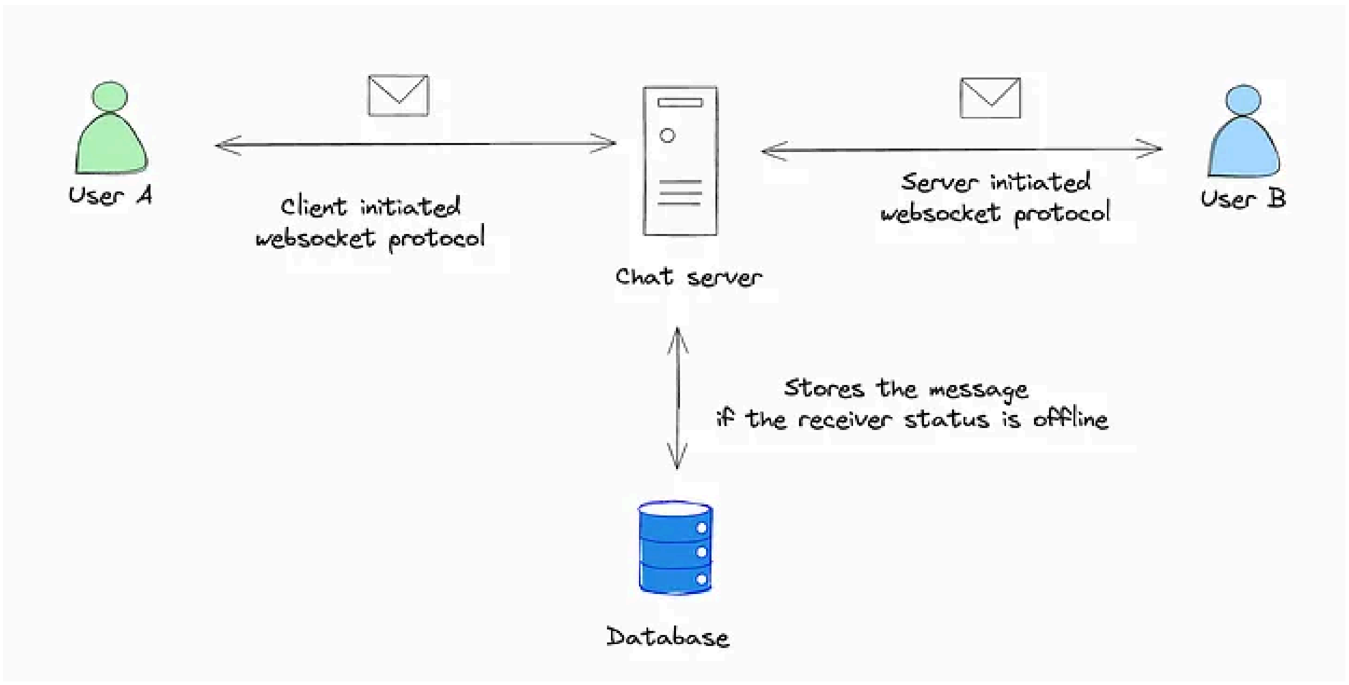

User A
Client initiated websocket protocol
Chat server
Server initiated websocket protocol
User B
Stores the message if the receiver status is offline
Database

Fig 1.0: High level design of Whatsapp

1. User A and user B create a communication channel with the chat server.
2. User A sends a message to the chat server.
3. Upon receiving the message, the chat server acknowledges back to user A.
4. The chat server sends the message to user B and stores the message in the database if the receiver's status is offline.

5. User B sends an acknowledgment to the chat server.
6. The chat server notifies user A that the message has been successfully delivered.
7. When user B reads the message, the application notifies the chat server.
8. The chat server notifies user A that user B has read the message.

Detailed design

Connection with a WebSocket server

In WhatsApp, each active device is connected with a **WebSocket server** via WebSocket protocol. A WebSocket server keeps the connection open with all the active (online) users. Since one server isn't enough to handle billions of devices, there should be enough servers to handle billions of users. The responsibility of each of these servers is to **provide a port to every online user**. The mapping between servers, ports, and users is stored in the **WebSocket manager** that resides on top of a cluster of the data store. In this case, that's **Redis**.

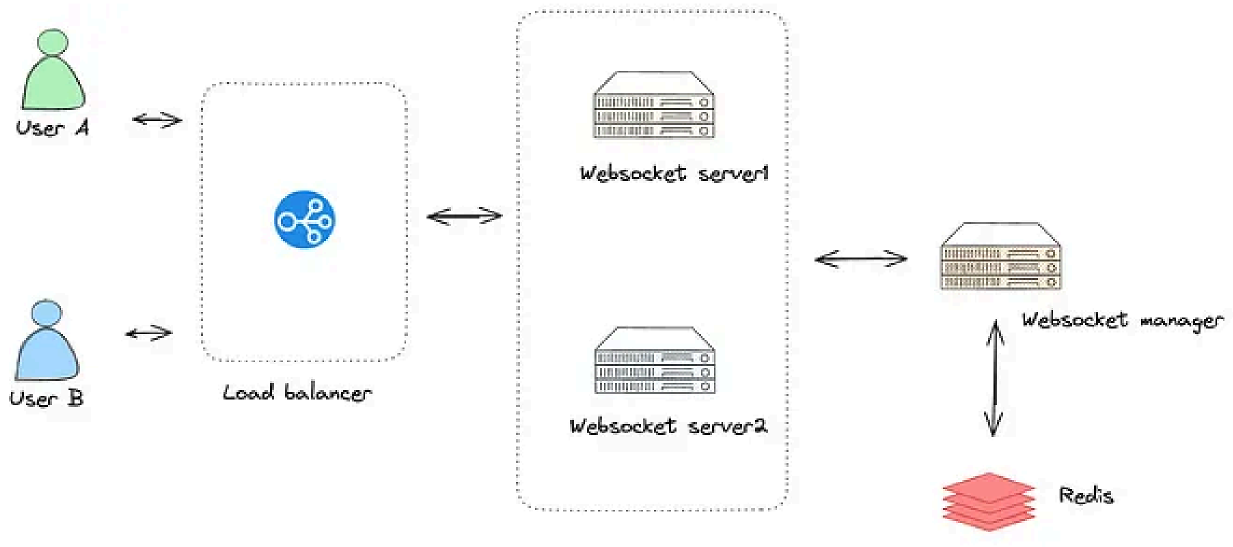


Fig 2.0: Connection with a distributed Websocket server

Send or receive messages

The WebSocket manager is responsible for:

- Maintaining a mapping between an active user and a port assigned to the user. Whenever a user is connected to another WebSocket server, this information will be updated in the data store. `userId<>server:port`

Both users are connected to the same server

If both users are connected to the same server, the call to the WebSocket manager is avoided.

Both users are connected to different servers

The WebSocket server associated with user A identifies the WebSocket to which user B is connected via the **WebSocket manager**. If user B is online, the WebSocket manager responds back to user A's WebSocket server that user B is connected with its WebSocket server.

Simultaneously, the WebSocket server sends the message to the message service and is stored in the **Mnesia** database where it gets processed in the first-in-first-out (FIFO) order. As soon as these messages are delivered to the receiver, they are deleted from the database.

If user B is offline, messages are kept in the Mnesia database. Whenever they become online, all the messages intended for user B are delivered via **push notification**. Otherwise, these messages are deleted permanently after **30 days.**

Both users (sender and receiver) communicate with the WebSocket manager to find each other's WebSocket server. Consider a case where there can be a continuous conversation between both users. This way, **many calls** are made to the WebSocket manager. To minimize the latency and reduce the number of these calls to the WebSocket manager, each WebSocket server caches information of recent conversations about which user is connected to which WebSocket server.

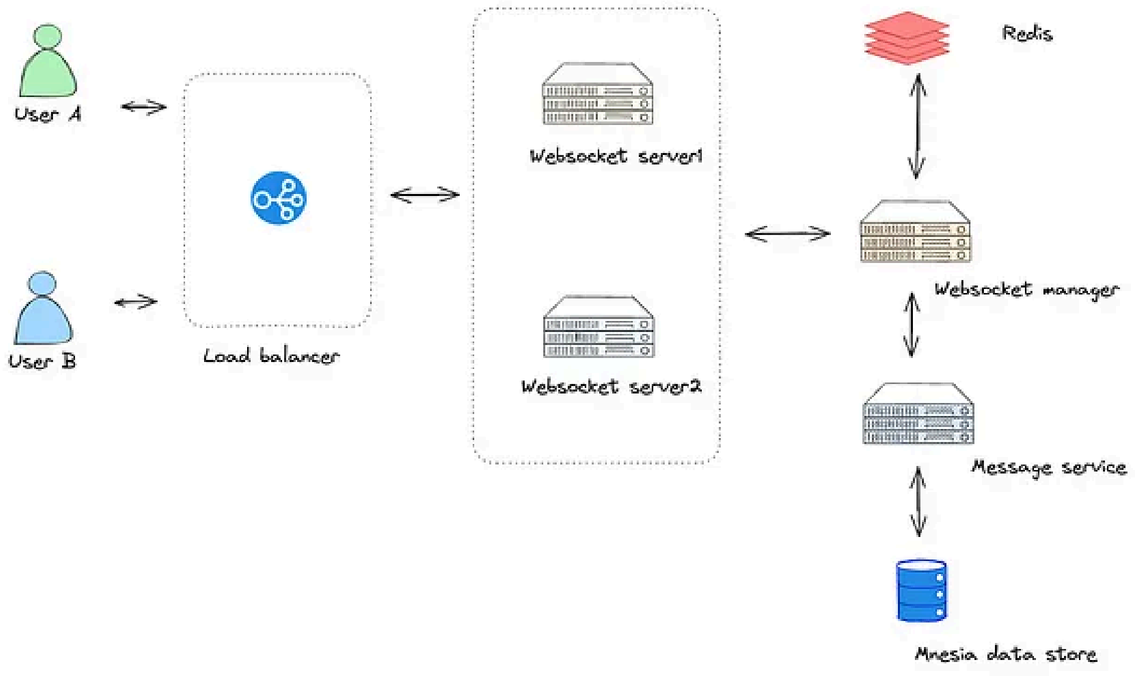


Fig 3.0: WebSocket communicates with message service on top of Mnesia database cluster

Send or receive media files

Usually, the **WebSocket servers** are lightweight and **don't support heavy logic** such as handling the sending and receiving of media files. We have another service called the **asset service**, which is responsible for sending and receiving media files.

1. The media file is compressed and encrypted on the device side.
2. The compressed and encrypted file is sent to the asset service to store the file on blob storage. The asset service assigns an ID that's communicated with the sender. The asset service also maintains a hash for each file to avoid duplication of content on the blob storage. For example, if a user wants to upload an image that's already there in the blob storage, the image won't be uploaded. Instead, the same ID is forwarded to the receiver.
3. The asset service sends the ID of media files to the receiver via the message service. The receiver downloads the media file from the blob storage using the ID.
4. The content is loaded onto a CDN if the asset service receives a large number of requests for some particular content.

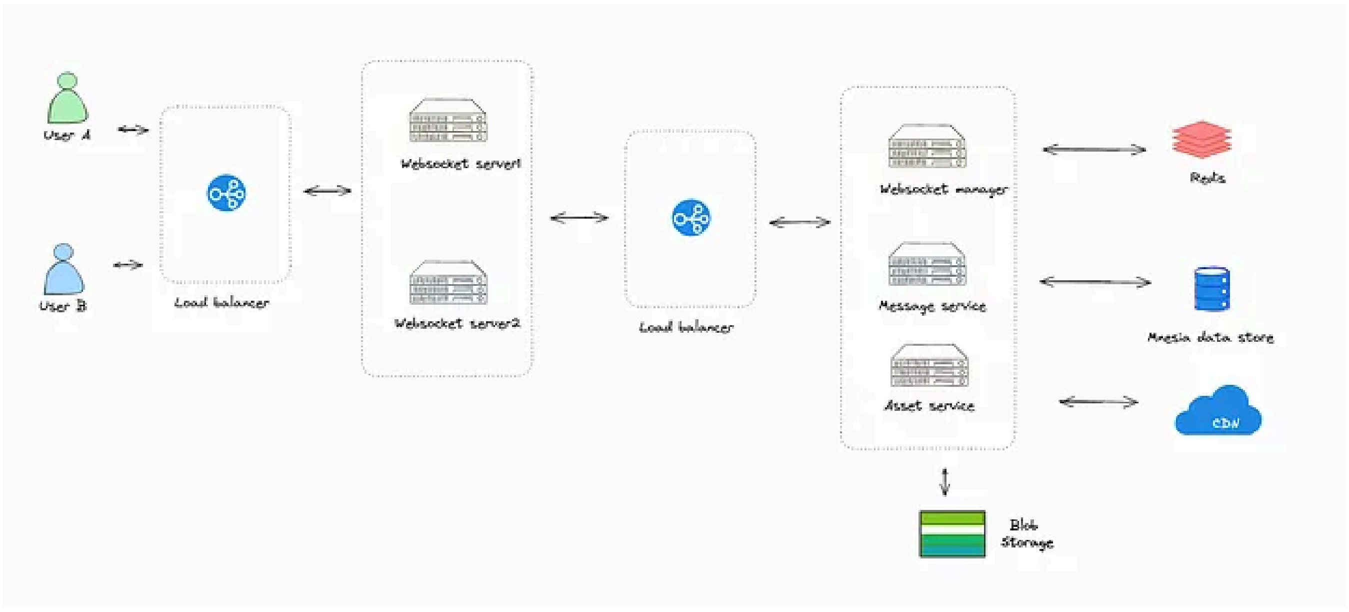


Fig 4.0: Sending media files via the asset service

Support for group messages

WebSocket servers don't keep track of groups because they only track active users. However, some users could be online and others could be offline in a group. For group messages, the following three main components are responsible for delivering messages to each user in a group:

- Group message handler
- Group message service
- Kafka

Let's assume that user A wants to send a message to a group with some unique ID — for example, `Group/A` . The following steps explain the flow of a message sent to a group:

1. Since user A is connected to a WebSocket server, it sends a message to the **message service** intended for `Group/A` .
2. The **message service sends the message to Kafka** with other specific information about the group. The message is saved there for further processing. In Kafka terminology, a group can be a topic, and the senders and receivers can be producers and consumers, respectively.
3. Now, here comes the responsibility of the group service. The **group service** keeps all information about users in each group in the system. It has all the **information about each group, including user IDs, group ID, status, group icon,**

number of users, and so on. This service resides on top of the **MySQL database** cluster, with multiple secondary replicas distributed geographically. A **Redis cache** server also exists to cache data from the MySQL servers. Both geographically distributed replicas and Redis cache aid in reducing latency.

4. The **group message handler** communicates with the **group service** to retrieve data of `Group/A` users.

5. In the last step, the group message handler follows the same process as a WebSocket server and delivers the message to each user.

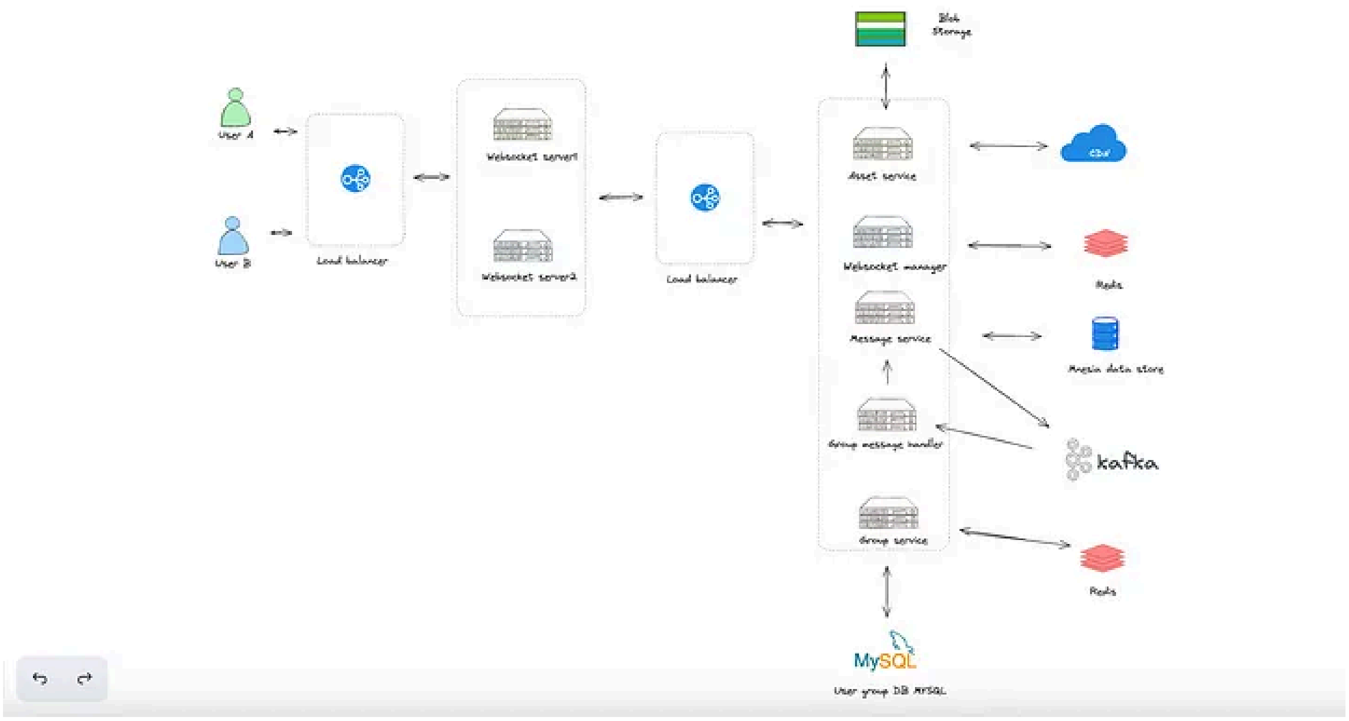


Fig 5.0: Components responsible for sending group messages

Evaluation

Low latency

We can minimize the latency of the system at various levels: We can do this through geographically distributed WebSocket servers and the cache associated with them. We can use Redis cache clusters on top of MySQL database clusters. We can use CDNs for frequently sharing documents and media content.

Consistency

The system also provides high consistency in messages with the help of a FIFO messaging queue with strict ordering. However, the ordering of messages would require the Sequencer to provide ID with appropriate causality inference mechanisms to each message. For offline users, the Mnesia database stores messages in a queue. The messages are sent later in a sequence after the user goes online.

Availability

The system can be made highly available if we have enough WebSocket servers and replicate data across multiple servers. When a user gets disconnected due to some fault in the WebSocket server, the session is re-created via a load balancer with a different server. Moreover, the messages are stored on the Mnesia cluster following the primary-secondary replication model, which provides high availability and durability.

Security

The system also provides an end-to-end encryption mechanism that secures the chat between users.

Summary

In this chapter, we designed a WhatsApp messenger. First, we identified the functional and non-functional requirements along with the resources estimation crucial for the design. Second, we focused on the high-level and detailed design of the WhatsApp system, where we described various components responsible for different services. Finally, we evaluated the non-functional requirements and highlighted some trade-offs in the design.

This design problem highlighted that we can optimize general-purpose computational resources for specific use cases. WhatsApp optimized its software stack to handle a substantially large number of connections on the commodity servers

WhatsApp System Design Interview Software Architect Software Architecture Distributed System Design

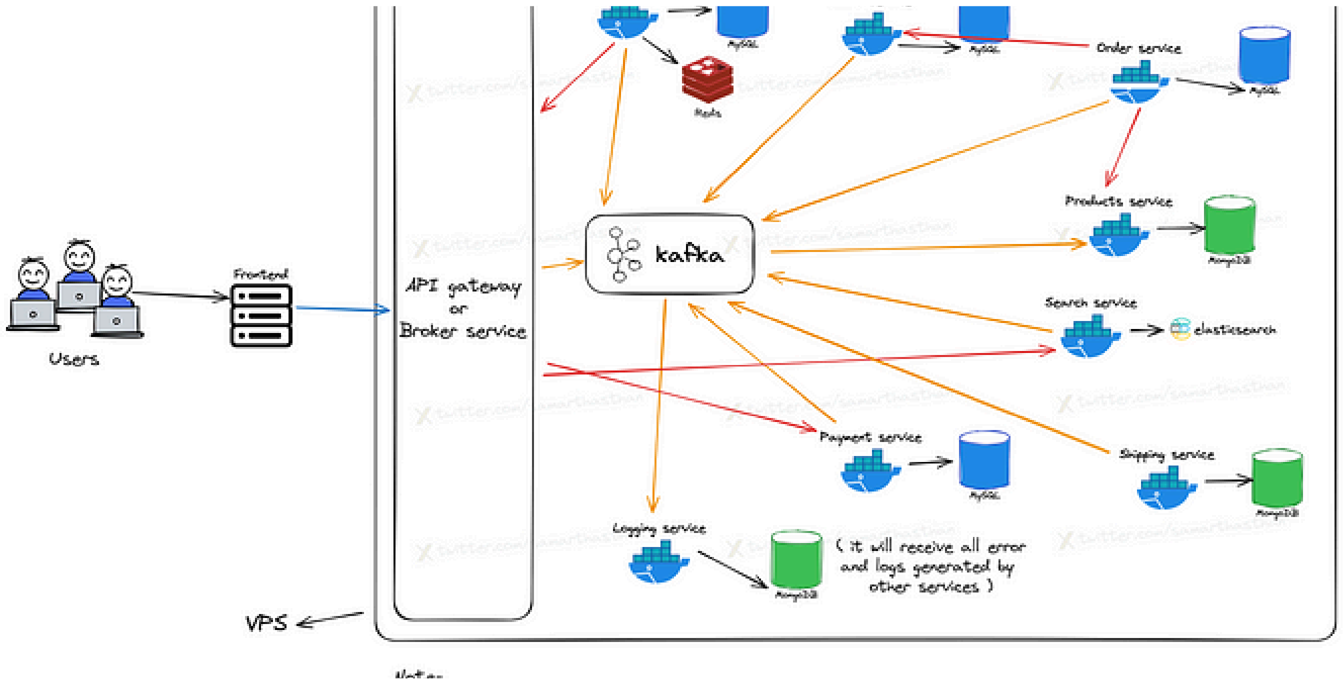
Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium



Samarth Asthan

Building a Scalable E-commerce Empire: A Micro-services System Design Approach

Hey everyone! I'm Samarth Asthan, a 3rd-year computer science student, and I'm new to the world of system design. But, I'm excited to share...

3 min read · Dec 11, 2023

8